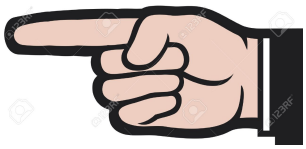
Endereços e ponteiros



Os conceitos de endereço e ponteiro são fundamentais em qualquer linguagem de programação, embora fiquem ocultos em algumas linguagens. Em C, esses conceitos são explícitos. Dominar o conceito de ponteiro exige algum esforço e uma boa dose de prática.

Endereços

A memória RAM de qualquer computador é uma sequência de bytes. Cada byte armazena um de 256 possíveis valores. Os bytes são numerados sequencialmente e o número de um byte é o seu *endereço* (= *address*).

Cada objeto na memória do computador ocupa um certo número de bytes consecutivos. Um `char` ocupa 1 byte. Um `int` ocupa 4 bytes e um `double` ocupa 8 bytes em muitos computadores. O número exato de bytes de um objeto é dado pelo operador **`sizeof`**: a expressão `sizeof(int)`, por exemplo, dá o número de bytes de um `int` no seu computador.

Cada objeto na memória tem um *endereço*. Na maioria dos computadores, o endereço de um objeto é o endereço do seu primeiro byte. Por exemplo, depois das declarações

```
char c;
int i;
struct {
   int x, y;
} ponto;
int v[4];
```

os endereços das variáveis poderiam ser os seguintes

```
c       89421
i       89422
ponto   89426
v[0]    89434
v[1]    89438
v[2]    89442
```

O endereço de um objeto (como uma variável, por exemplo) é dado pelo operador **`&`**. Se **`i`** é uma variável então

`&i`

é o seu endereço. (Não confunda esse uso de "`&`" com o operador lógico *and*, que se escreve "`&&`" em C.) No exemplo acima, `&i` vale *89422* e `&v[3]` vale *89446*.

Um exemplo: O segundo argumento da função de biblioteca `scanf` é o endereço da variável onde deve ser depositado o objeto lido do dispositivo padrão de entrada:

```
int i;
scanf ("%d", &i);
```

Exercícios 1

1. TAMANHOS. Compile e execute o seguinte programa:

```
int main (void) {
   typedef struct {
      int dia, mes, ano;
   } data;
   printf ("sizeof (data) = %d\n",
            sizeof (data));
}
```

Ponteiros

Um *ponteiro* (= apontador = *pointer*) é um tipo especial de variável que armazena endereços. Um ponteiro pode ter o valor especial

`NULL`

que não é endereço de lugar algum. A macro `NULL` está definida na interface `stdlib.h` e seu valor é `0` na maioria dos computadores.

Se um ponteiro `p` armazena o endereço de uma variável `i`, podemos dizer "`p` aponta para `i`" ou "`p` é o endereço de `i`". (Em termos um pouco mais abstratos, diz-se que `p` é uma *referência* à variável `i`.) Se um ponteiro `p` tem valor diferente de `NULL` então

`*p`

é o *valor* do objeto apontado por `p`. (Não confunda esse uso de "`*`" com o operador de multiplicação!) Por exemplo, se `i` é uma variável e `p` vale `&i` então dizer "`*p`" é o mesmo que dizer "`i`".

Figura esquerda: um ponteiro `p`, armazenado no endereço *60001*, contém o endereço de um inteiro. Figura direita: representação esquemática da situação.

Há vários tipos de ponteiros: ponteiros para bytes, ponteiros para inteiros, ponteiros para ponteiros para inteiros, ponteiros para registros etc. O computador precisa saber de que tipo de ponteiro você está falando. Para declarar um ponteiro `p` para um inteiro, diga

```
int *p;
```

(Há quem prefira `int* p`.) Para declarar um ponteiro `p` para um registro `reg`, diga

```
struct reg *p;
```

Um ponteiro `r` para um ponteiro que apontará um inteiro é declarado assim:

```
int **r;
```

(Veja, por exemplo, a declaração de uma matriz de números inteiros.)

Exemplos. Suponha que `a`, `b` e `c` são variáveis inteiras e veja um jeito bobo de fazer "`c = a+b`":

```
int *p;  // p é um ponteiro para um inteiro
int *q;
p = &a;  // o valor de p é o endereço de a
q = &b;  // q aponta para b
c = *p + *q;
```

Outro exemplo bobo:

```
int *p;
int **r; // r é um ponteiro para ponteiro para inteiro
p = &a;  // p aponta para a
r = &p;  // r aponta para p e *r aponta para a
c = **r + b;
```

Aplicação

Suponha que precisamos de uma função que troque os valores de duas variáveis inteiras, digamos `i` e `j`. É claro que a função

```
void troca (int i, int j) { // errado!
   int temp;
   temp = i; i = j; j = temp;
}
```

não produz o efeito desejado, pois recebe apenas os *valores* das variáveis e não as variáveis propriamente ditas. A função recebe "cópias" das variáveis e troca os valores dessas cópias, enquanto as variáveis "originais" permanecem inalteradas. Para obter o efeito desejado, é preciso passar à função os *endereços* das variáveis:

```
void troca (int *p, int *q)
{
   int temp;
   temp = *p; *p = *q; *q = temp;
}
```

Para aplicar essa função às variáveis `i` e `j` basta dizer

```
troca (&i, &j);
```

ou talvez

```
int *p, *q;
p = &i; q = &j;
troca (p, q);
```

Exercícios 2

1. Verifique que a troca de valores de variáveis discutida acima poderia ser obtida por meio de uma macro do pré-processador:

   ```
   #define troca (X, Y) { int t = X; X = Y; Y = t; }
   . . .
   troca (i, j);
   ```

2. Por que o código abaixo está errado?

   ```
   void troca (int *i, int *j) {
      int *temp;
      *temp = *i; *i = *j; *j = *temp;
   }
   ```

3. Um ponteiro pode ser usado para dizer a uma função onde ela deve depositar o resultado de seus cálculos. Escreva uma função `hm` que converta minutos em horas-e-minutos. A função recebe um inteiro `mnts` e os endereços de duas variáveis inteiras, digamos `h` e `m`, e atribui valores a essas variáveis de modo que `m` seja menor que 60 e que `60*h + m` seja igual a `mnts`. Escreva também uma função `main` que use a função `hm`.

4. Escreva uma função `mm` que receba um vetor inteiro `v[0..n-1]` e os endereços de duas variáveis inteiras, digamos `min` e `max`, e deposite nessas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Escreva também uma função `main` que use a função `mm`.

Vetores e endereços

Os elementos de qualquer vetor (= *array*) têm endereços *consecutivos* na memória do computador. (Na verdade, os endereços não são consecutivos, uma vez que cada elemento do vetor pode ocupar vários bytes. Mas o compilador C acerta os detalhes internos de modo a criar a ilusão de que a diferença entre os endereços de elementos consecutivos vale 1.) Por exemplo, depois da declaração

```
int *v;
v = malloc (100 * sizeof (int));
```

o ponteiro `v` aponta o primeiro elemento de um vetor de 100 elementos. O endereço do segundo elemento do vetor é `v+1` e o endereço do terceiro elemento é `v+2`. Se `i` é uma variável do tipo `int` então

`v + i`

é o endereço do `(i+1)`-ésimo elemento do vetor. As expressões `v + i` e `&v[i]` têm exatamente o mesmo valor e portanto as atribuições

```
*(v+i) = 87;
v[i] = 87;
```

têm o mesmo efeito. Analogamente, qualquer dos dois fragmentos de código abaixo pode ser usado para preencher o vetor `v`:

```
for (i = 0; i < 100; ++i)  scanf ("%d", &v[i]);
for (i = 0; i < 100; ++i)  scanf ("%d", v + i);
```

Todas essas considerações também valem se o vetor for alocado estaticamente pela declaração

```
int v[100];
```

mas nesse caso `v` é uma espécie de "ponteiro constante", cujo valor não pode ser alterado.

Exercícios 3

1. Suponha que os elementos de um vetor `v` são do tipo `int` e cada `int` ocupa 8 bytes no seu computador. Se o endereço de `v[0]` é `55000`, qual o valor da expressão `v + 3`?
2. Suponha que `v` é um vetor declarado assim:

   ```
   int v[100];
   ```

   Descreva, em português, a sequência de operações que deve ser executada para calcular o valor da expressão `&v[k + 9]`.
3. Suponha que `v` é um vetor. Descreva a diferença conceitual entre as expressões `v[3]` e `v + 3`.
4. O que faz a seguinte função?

   ```
   void imprime (char *v, int n) {
      char *c;
      for (c = v; c < v + n; v++)
         printf ("%c", *c);
   }
   ```
5. O seguinte fragmento de código pretende decidir se "abacate" vem antes ou depois de "uva" no dicionário. O que há de errado?

   ```
   char *a, *b;
   a = "abacate"; b = "uva";
   if (a < b)
      printf ("%s vem antes de %s\n", a, b);
   else
      printf ("%s vem depois de %s\n", a, b);
   ```

Veja o verbete *Pointer (computer programming)* na Wikipedia

Aula em vídeo sobre ponteiros no Academic Earth (usa C++, mas os conceitos são os mesmos de C)
Aula em vídeo sobre aritmética de ponteiros na The Open Academy

---

Atualizado em 2018-01-16
https://www.ime.usp.br/~pf/algoritmos/